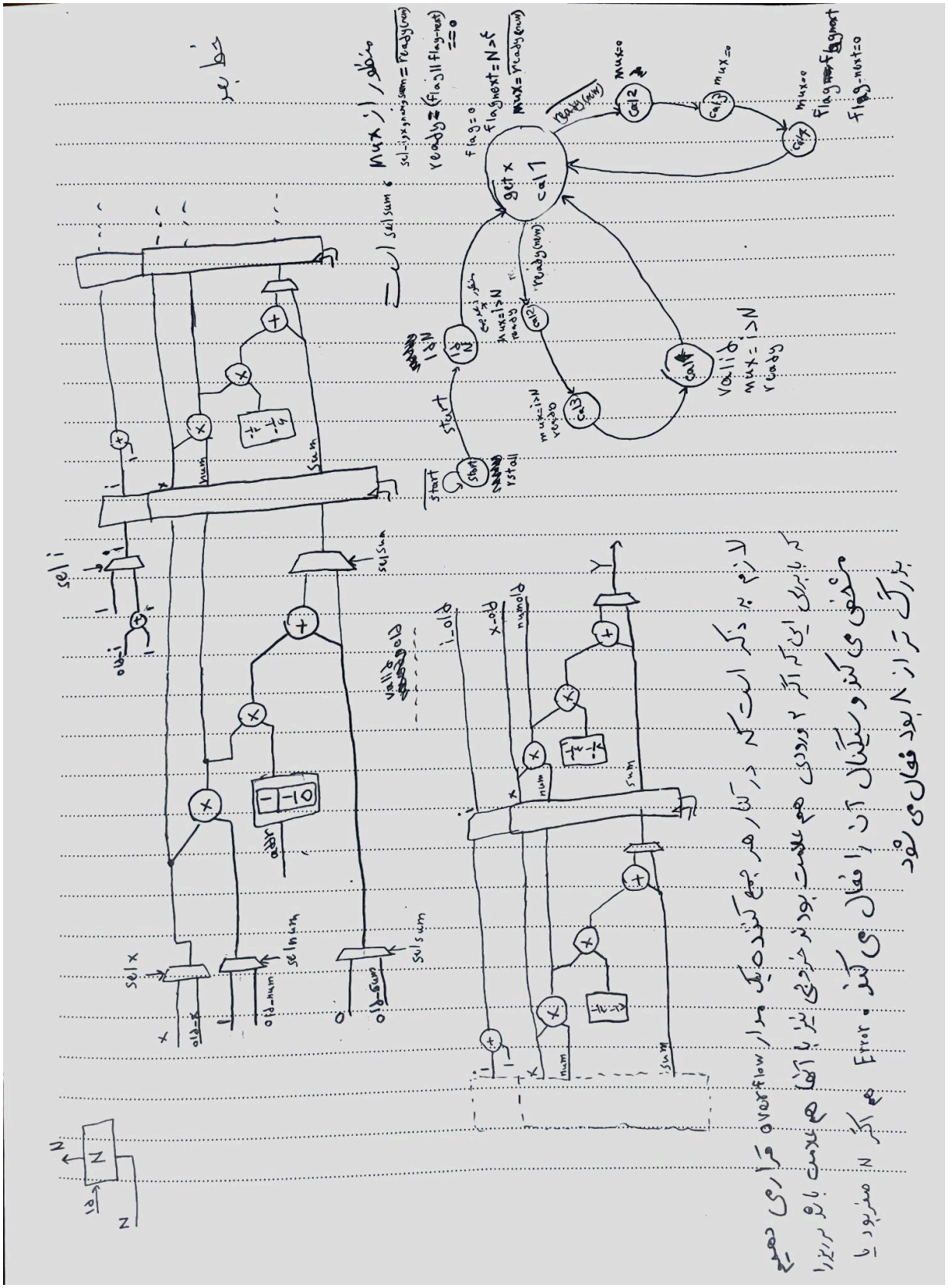
بسم الله الرحمن الرحیم

میانترم درس طراحی کامپیوتری سیستم های دیجیتال

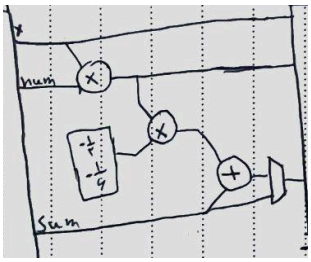
دکتر صالحی

مهدی وجهی - ۸۱۰۱۰۱۵۵۸

کنترلر و مسیر داده



اسلایس های مسیر داده پایپ



هر اسلایس مسیر داده پایپ به صورت روبه رو طراحی شده که موارد زیر را ورودی می گیرد:

- X: عددی که در سری باید قرار داده شود.
- Num: نشان دهنده X به توان n هست که در هر مرحله با توجه به افزایش n مجدد در x ضرب می شود و به اسلایس بعد منتقل می شود.
- Addr: مشخص می کند که داده اول رم روی مدار قرار گیرد یا دومی
- Sum: مجموع عبارت های سری تا این لحظه
- selSum: مشخص می کند که مقدار قبلی sum منتقل شود یا مقدار جدید آن.
- Overflow: جهت انتقال سریز از بخش های قبل

مدار همین موارد را برای اسلایس بعدی نیز بروزرسانی می کند. برای این کار num جدید باید با ضرب num ورودی در x تولید شود و sum جدید هم با جمع شدن sum قبلی با num*coefficient محاسبه می شود همچنین اگر مقدار N تعیین شده به صورتی باشد که لازم نباشد جملات بعدی با آن جمع شود صرفا کافیست selSum مقدار قبلی را روی خروجی بگذارد. همانطور که در برگه شرح دادیم باید سرریز را هم شناسایی کنیم که این موضوع را با یک بیت خروجی overflow تشخیص می دهیم لازم است این مقدار را یا از طریق کنترلر یا مسیر داده منتقل کنیم که در کد حالت دوم را در نظر گرفتیم.

پیاده سازی

```
module pipe_slice_dp #(
   parameter RAM_VALUE_0 = 32'b0,
   parameter RAM_VALUE_1 = 32'b0
) (
   input signed  [31:0] in_x,
   input signed  [31:0] in_num,
   input signed  [31:0] in_sum,
   input         addr,
   input         sel_sum,
   input         in_overflow,
   output signed [31:0] out_sum,
   output signed [31:0] out_num,
   output signed [31:0] out_x,
   output        out_overflow
);

   wire signed [31:0] term;
   wire signed [31:0] add;
   wire signed [31:0] coefficient;
```

```
    wire signed [63:0] mult1;
    wire signed [63:0] mult2;
    wire        overflow_flag;

    assign out_x = in_x;

    assign mult1 = in_x * in_num;
    assign out_num = mult1[62:31];

    assign mult2 = coefficient * out_num;
    assign term = mult2[62:31];

    adder adder_instance (
        .a(term),
        .b(in_sum),
        .sum(add),
        .overflow(overflow_flag)
    );
    assign out_overflow = in_overflow | overflow_flag;

    assign out_sum = sel_sum ? in_sum : add;

    ram #(
        .RAM_VALUE_0(RAM_VALUE_0),
        .RAM_VALUE_1(RAM_VALUE_1)
    ) ram_instance (
        .addr(addr),
        .data_out(coefficient)
    );


endmodule
```

همانطور که می بینید مقادیر رم به صورت پارامتر دریافت شده تا به رم داخلی پاس دهد. همان مواردی که گفته شد در کد پیاده شده. برای ضرب ها چون ضرب ۲ عدد ۳۲ بیت هست و اعشاری ۳۲ بیت با ارزش برداشته می شود. کد مدار جمع هم به صورت زیر است که مطابق توضیحات هست.

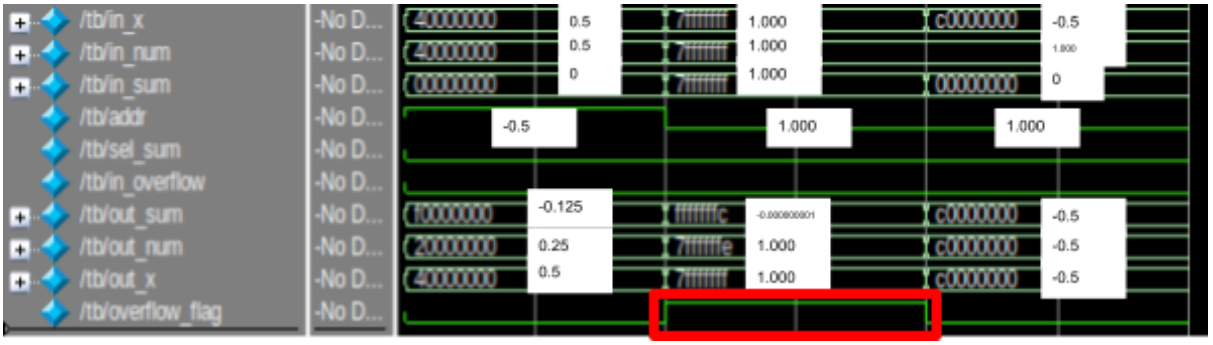
```
module adder (
    input  signed [31:0] a,
    input  signed [31:0] b,
    output signed [31:0] sum,
    output        overflow
);
```

```
   assign sum = a + b;

   assign overflow = ((a[31] == b[31]) && (sum[31] != a[31]));

endmodule
```

مدار رم هم توضیحاتی ندارد و می توانید در ram.v مشاهده کنید.

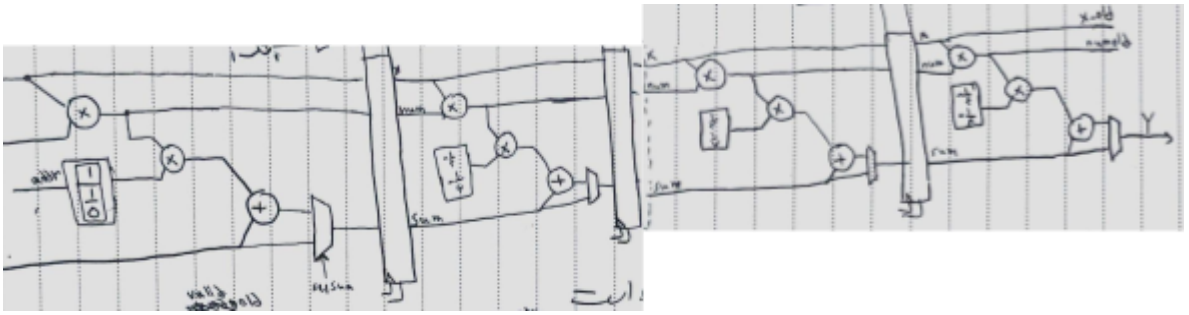
تست بنچ

Tb_slice.v را برای آن می نویسیم نتیجه به صورت زیر است که درستی مدار را نشان می دهد.



مسیر داده پایپ ۴ تایی

حال مسیر داده ۴ مرحله ای را مطابق طراحی پیدا سازی می کنیم. اسلایس ها طراحی شده اند. ماژول های رجیستر هم چون نکته خاصی ندارند از شرح آنها صرف نظر می کنیم (می توانید در dp_reg.v) مشاهده کنید.



تنها کافیست که آنها را به هم وصل کنیم که بخشی از کد نتیجه را می توانید مشاهده کنید.

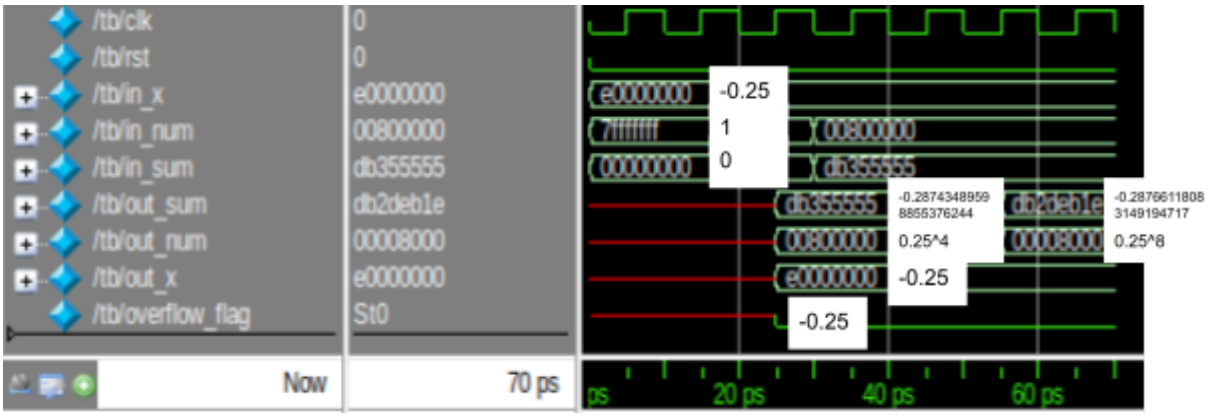
```
module pipe_4_stage #(
   parameter RAM_VALUE_0_STAGE1 = 32'h7FFFFFFF,
   parameter RAM_VALUE_1_STAGE1 = 32'h19999999,
   parameter RAM_VALUE_0_STAGE2 = 32'hC0000000,
   parameter RAM_VALUE_1_STAGE2 = 32'hF5555555,
) (
```

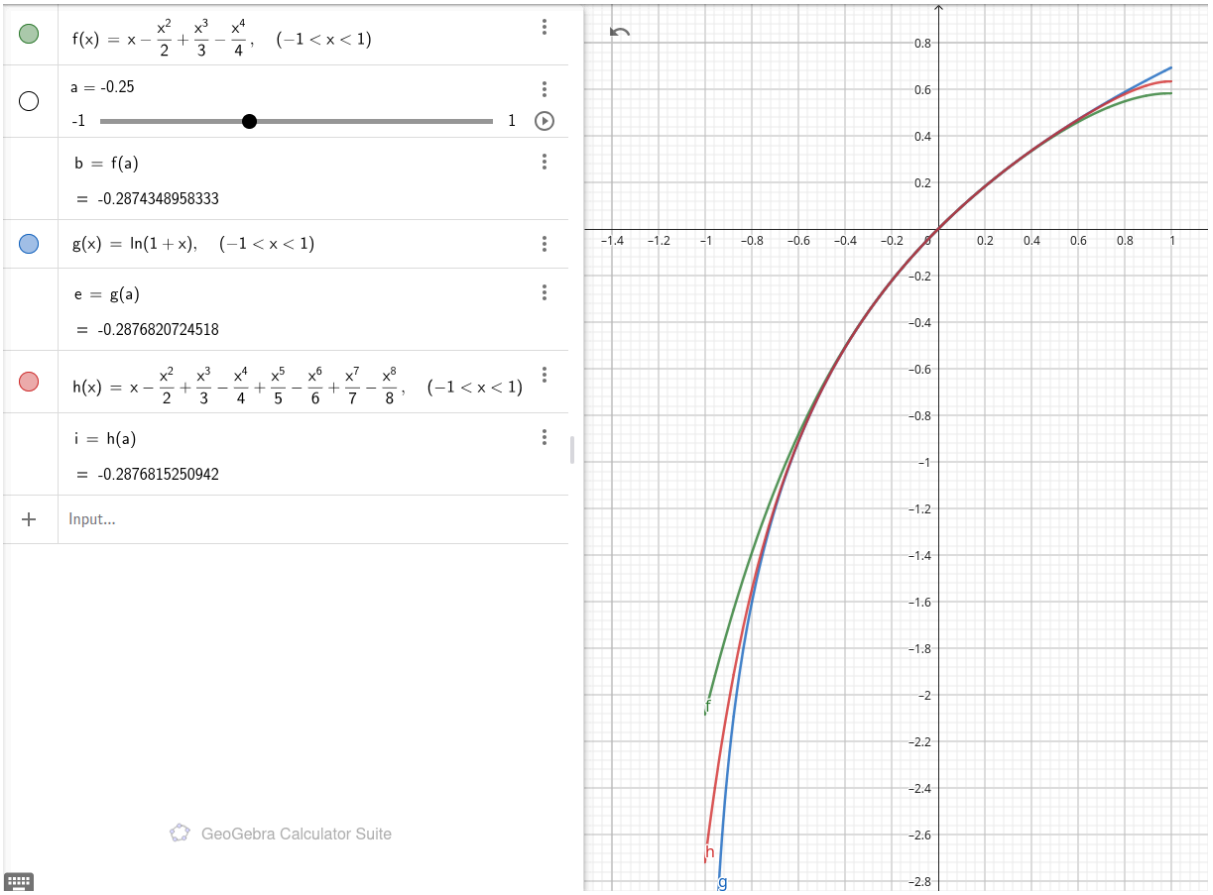
```
    input          clk,
    input          rst,
    input  [31:0] in_x,
    input  [31:0] in_num,
    input  [31:0] in_sum,
    input          addr_stage1, addr_stage2, addr_stage3, addr_stage4,
    input          sel_sum_stage1, sel_sum_stage2, sel_sum_stage3,
sel_sum_stage4,
    input          in_overflow,
    output [31:0] out_sum,
    output [31:0] out_num,
    output [31:0] out_x,
    output         overflow_out
);

    pipe_slice_dp #(
        .RAM_VALUE_0(RAM_VALUE_0_STAGE1),
        .RAM_VALUE_1(RAM_VALUE_1_STAGE1)
    ) stage1 (
        .in_x(in_x),
        .in_num(in_num),
        .in_sum(in_sum),
        .addr(addr_stage1),
        .sel_sum(sel_sum_stage1),
        .in_overflow(in_overflow),
        .out_sum(stage1_out_sum),
        .out_num(stage1_out_num),
        .out_x(stage1_out_x),
        .out_overflow(overflow_out_stage1)
    );

    dp_reg S1to2 (
        .clk(clk),
        .rst(rst),
        .in_x(stage1_out_x),
        .in_num(stage1_out_num),
        .in_sum(stage1_out_sum),
        .in_overflow(overflow_out_stage1),
        .out_x(stage1_reg_x),
        .out_num(stage1_reg_num),
        .out_sum(stage1_reg_sum),
        .out_overflow(stage1_reg_overflow)
    );
```

```
  pipe_slice_dp #(
      .RAM_VALUE_0(RAM_VALUE_0_STAGE2),
      .RAM_VALUE_1(RAM_VALUE_1_STAGE2)
  ) stage2 (
      .in_x(stage1_reg_x),
      .in_num(stage1_reg_num),
      .in_sum(stage1_reg_sum),
      .addr(addr_stage2),
      .sel_sum(sel_sum_stage2),
      .in_overflow(stage1_reg_overflow),
      .out_sum(stage2_out_sum),
      .out_num(stage2_out_num),
      .out_x(stage2_out_x),
      .out_overflow(overflow_out_stage2)
  );
...
```

تست بنچ

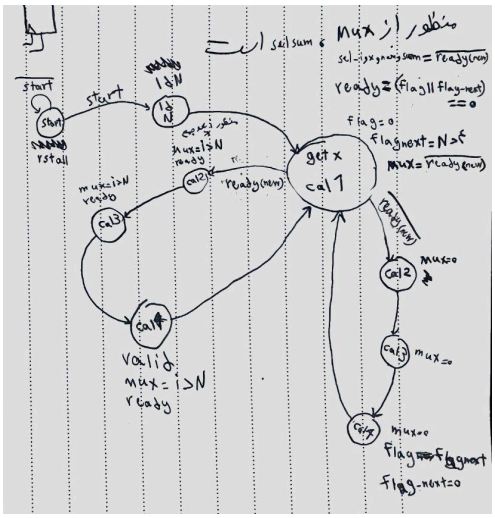
Tb_pipe.v را برای آن می نویسیم نتیجه به صورت زیر است. ابتدا مقدار منفی ۰.۵ را برای x وارد می کنیم و بعد از ۳۰ واحد که مقدار روی خروجی آمد مجدد بر می گردانیم که مقدار دقیق تر (با ۸ مرحله) محاسبه شود.





خروجی با جواب مسئله تطابق دارد.

کنترلر



کنترلر طراحی شده به صورت روبه رو است که یک مقداری مفهومی است و کمی با پیاده سازی تفاوت می کند که در ادامه توضیح داده می شود. در ابتدا ما منتظر سیگنال شروع می مانیم. با ارسال آن مقدار N را ذخیره می کنیم و حال به دریافت و محاسبه مقادیر می پردازیم. اگر مقدار کمتر از ۵ باشد یعنی می توانیم بدون توقف داده بگیریم و اگر توجه کنید در استیت های محاسبه سمت چپ این موضوع گفته ذکر شده و سیگنال ready فعال می ماند. اما اگر N بزرگتر از ۴ بود باید بعد از دریافت ۴ داده منتظر بمانیم که داده ها با برگشت مجدد به پاپ محاسبه شوند. اما در امتحان من کلا پایپ را مسدود می کردم تا مقدار آن محاسبه شود و سپس ورودی جدید می گرفتم. (که بعد امتحان متوجه شدم که اینطور نبود) همچنین همانطور که در مسیر داده رسم شده بود سیگنال i را برای شمارش استفاده می کنیم.

تغییرات

تغییرات انجام شده به شرح زیر است:

- حذف استیت لود N و ذخیره N به عنوان پارامتر <u>با توجه به گفته استاد در جلسه که در قسمت پیاده سازی N را به صورت پارامتر دریافت کنیم.</u>
- تبدیل استیت های محاسبه به سیگنال و تبدیل ۷ استیت آن به ۱ استیت برای استفاده در پایپ. (۴ بخش محاسبه در ۴ پایپ حساب شده و کنترلر به جای پیمایش ۴ استیت سیگنال آن را تولید می کند)
- تغییر نام flag_next به big_N و تغییر نام flag به flag_next. (صرفا برای جلوگیری از ابهام گفتم)
- اضافه کردن سیگنال ready و valid به استیت شروع
- جابجایی جمع کننده و مالتی پلکسر i برای یکسان شدن پایپ ها

جزئیات اضافه شده

تغییرات اضافه شده به شرح زیر است:

- افزودن امکان گرفتن ۴ داده پشت سر هم برای N های بزرگتر از ۴ (تغییر شرط valid به سبب آن)
- افزودن sel_overflow برای انتقال سریز قسمت قبل برای N ها بزرگ تر از ۴
- با توجه به این که در مسیر سمت راست mux=0 است و همچنین همواره i<N است جهت کاهش پیچیدگی کلا از شرط mux= i>=N استفاده شد.

پیاده سازی

ابتدا ورودی ها را تنظیم می کنیم.

```
module controller #(parameter N = 3'b111) (
   input        clk,
   input        rst,
   input        start,
   input        in_flag_next,
   output reg      ready,
   output reg      out_valid,
   output reg      error,
   output reg      sel_x,
   output reg      sel_num,
   output reg      sel_sum,
   output reg      sel_i,
   output reg      sel_overflow,
   output reg      out_flag_next
);
```

سپس طبق توضیحات استیت ها را تنظیم می کنیم.

```
   parameter IDLE = 1'b0;
   parameter COMPUTE = 1'b1;

   wire big_N = N > 3'd3;

   reg state, next_state;

   always @(posedge clk , posedge rst) begin
       if (rst) begin
           state <= IDLE;
       end else begin
           state <= next_state;
       end
   end

   always @(*) begin
       next_state = state;
       case (state)
           IDLE: begin
               next_state = start ? COMPUTE : IDLE;
           end
           COMPUTE: begin
               next_state = COMPUTE;
```

```
            end
        endcase
    end
```

سپس مشابه طرح و تغییرات داده شده سیگنال های خروجی را تولید می کنیم.

```
    always @(*) begin
        {ready, error, out_valid} = 3'b000;
        case (state)
            IDLE: begin
                ready = start;
                out_valid = start && !big_N;
            end
            COMPUTE: begin
                ready = !in_flag_next || !big_N;
                out_valid = in_flag_next || !big_N;
            end
        endcase
    end

    assign out_flag_next = ready;
    assign sel_x = !ready;
    assign sel_num = !ready;
    assign sel_sum = !ready;
    assign sel_i = !ready;
    assign sel_overflow = !ready;

    assign error = 1'b0;
```

لازم به ذکر است که پیاده سازی i درون پایپ ها اضافه شد و مقادیری مانند next_flag در پایپ منتقل می شوند تا در هر مرحله مورد استفاده قرار گیرند همچنین مواردی که قبلا بدون کنترلر تنظیم می شد با اضافه شدن کنترلر یا توسط سیگنال های ارسالی آن یا به صورت مستقیم تنظیم می شود.

هنگامی سیگنال آماده فعال می شود که یا در ابتدا سیگنال شروع داده شود یا در N کمتر از مساوی ۴ باشد یا برای موارد بزرگ تر در دور اول محاسبات باشند و در دور بعدی ورودی جدید گرفته نمی شود. سیگنال درسیتی خروجی هم اگر داده کمتر از ۵ باشد کلا فعال به پایپ ارسال می شود و در غیر این صورت در دور دوم فعال می شود.

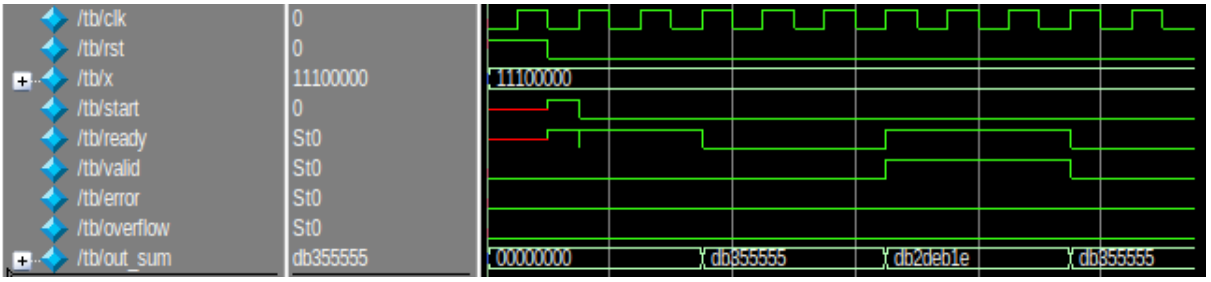
سیگنال های i اضافه شده به پایت برای کنترل:

```
    assign out_i = in_i + 1'b1;
    assign addr = in_i[2];
    assign sel_sum = in_i > N;
```

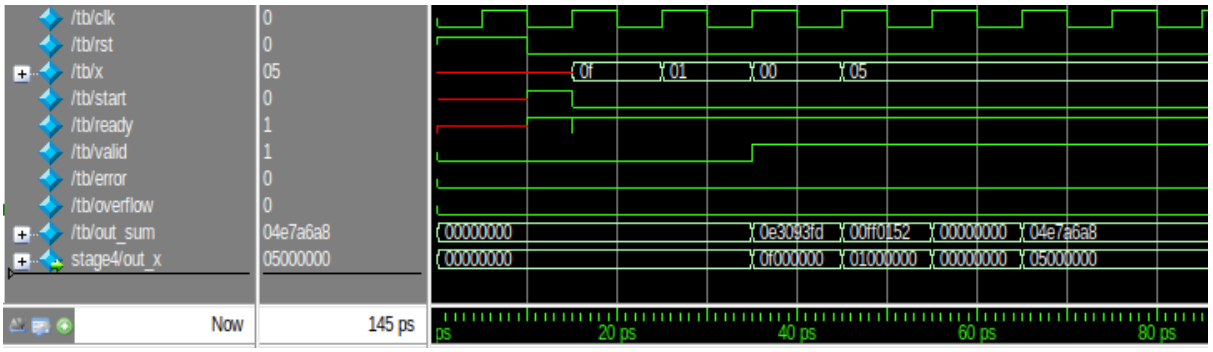
ما در دور دوم باید آدرس دوم را بخوانیم پس فقط وقتی N بزرگ تر از ۳ می شود خانه پایین انتخاب می شود. همچنین شرط sel_sum هم واضح است و توضیح داده شده.

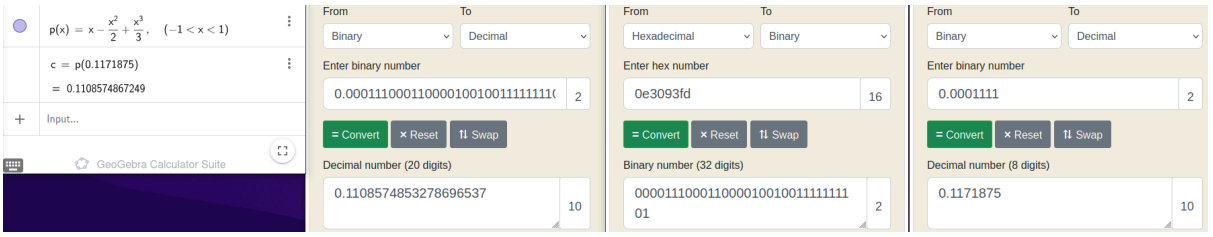
تست بنچ

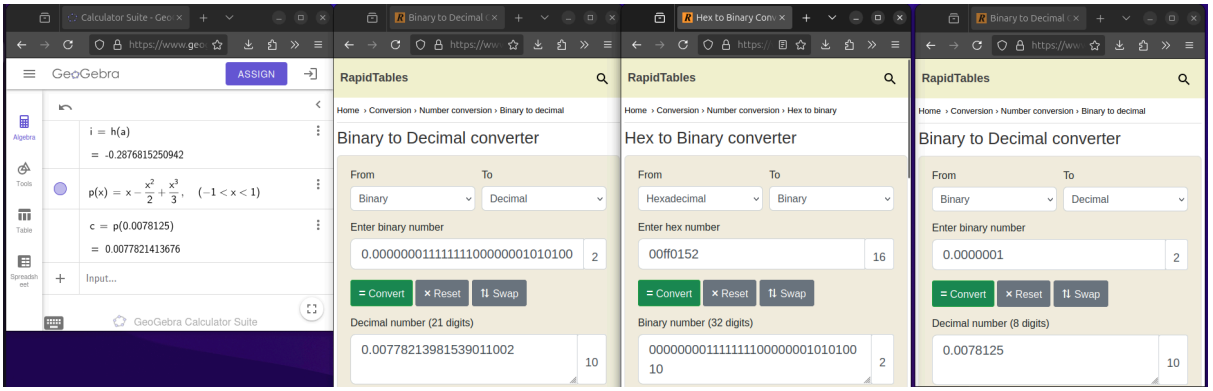
به ازای N=8 مانند قبل درست است



حال برای N=3 حساب می کنیم

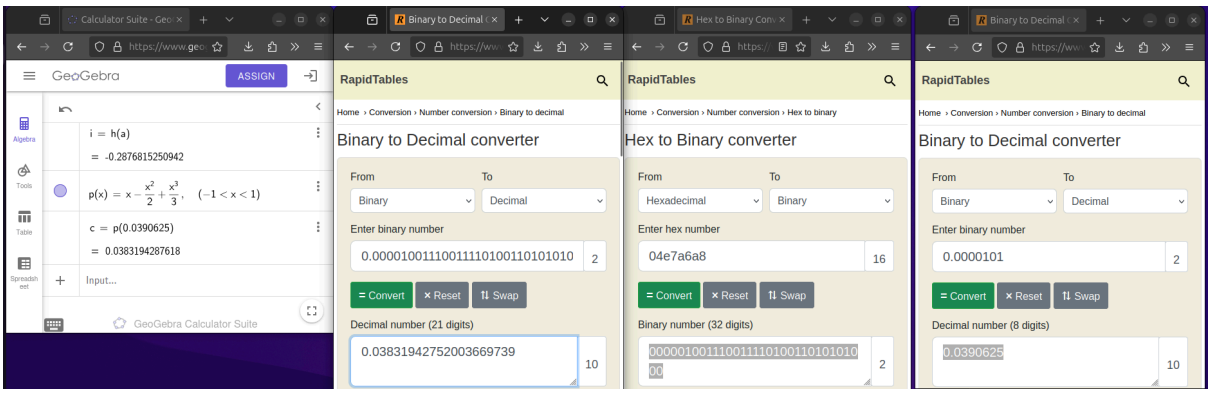






سومی واضحا صفر است.

13 | میانترم درس طراحی کامپیوتری سیستم های دیجیتال



برای N=5